MySQL

B+树索引的查询过程？为什么不用哈希索引？

B+树索引的查询过程：​

从**根**节点自上而下逐层比较确定下一层子节点的位置，最终定位到**叶子**节点

如果是范围查询，叶子节点通过链表连接，可以高效地遍历满足条件的数据。

为什么不用哈希索引？

B+树**有序 范围查询**、哈希索引**随机分布 等值查询**

分库分表后全局ID生成方案有哪些？

1. ​UUID：无序。  
   String uuid = UUID.randomUUID().toString();  
   Generated UUID: e3e70682-f3a1-11e9-8f0b-362b9e155667
2. ​Snowflake：时间戳 + 机器ID + 序列号。 时钟回拨可能引发问题。  
   0 | 41-bit 时间戳 | 10-bit 机器 ID | 12-bit 序列号
3. 数据库自增ID：依赖数据库。  
   
4. ​Redis自增ID：存在单点故障风险。  
   
5. ​号段模式​：从数据库预分配一批ID号段，本地缓存使用。号段用尽需重新申请，可能短暂阻塞。

主从同步延迟的解决方案？



事务隔离级别和MVCC的实现原理？

**事务隔离级别：​**

1. ​读未提交：最低级别
2. ​读已提交：解决脏读
3. ​可重复读​：默认级别，解决不可重复读
4. ​串行化：最高级别，解决幻读

**MVCC实现原理：​**

1. ​根据事务ID和版本时间戳判断数据可见性
2. ​写操作生成新版本，旧版本仍可被读取
3. ​定期清理旧数据，释放空间

死锁的产生条件及排查方法

**死锁产生四大条件**：​

1. ​互斥：资源一次只能被一个事务占用。
2. ​持等：事务持有资源的同时，等待其他资源。
3. ​不夺：已分配的资源不能被强制剥夺，只能由持有者释放。
4. ​循等：事务之间形成资源等待的环路。

**排查方法：​**

1. ​SHOW ENGINE INNODB STATUS;
2. ​查看LATEST DETECTED DEADLOCK：

* 查看涉事务ID
* 分析事务持有的锁和等待的锁
* 定位造成循环等待的SQL语句

1. ​调整锁顺序、事务粒度、重试机制

覆盖索引和索引下推的区别？

覆盖索引**：​**查询字段都包含在索引中，无需回表

索引下推**：​**在引擎层利用索引过滤数据，减少回表，适用于WHERE包含索引和非索引的查询



Redo Log和Undo Log的作用及写入机制？

**Redo Log**持久性、崩溃恢复：将**修改**写Buffer刷**磁盘**

**Undo Log**原子性、MVCC读一致性、回滚：将**旧数据**写**段**中刷磁盘

Change Buffer对写操作的优化原理？

如果数据库直接修改磁盘上的索引页，那么需要：将索引页从磁盘加载到内存，在内存中修改数据，最后写回磁盘。

当修改非唯一 二级索引时，先将修改记录到Change Buffer中，当相关索引页被加载到内存时，再将Change Buffer中的修改合并到索引页。

适用于写多读少

Redis

Redis单线程模型为什么能高效处理请求？

1. ​**内存**
2. ​**跳表**
3. **单线程 事件驱动模型 + I/O多路复用**

缓存穿透/击穿/雪崩的解决方案？

缓存穿透不存在：查询一个不存在的数据，缓存中没有相应的记录，每次请求都会去数据库查询，造成数据库负载激增。

* 布隆过滤器
* 缓存不存在标识

Redis缓存击穿热点：某个热点数据在缓存中过期，导致大量请求同时访问数据库，造成数据库负载激增。

* 互斥锁
* 热点数据永不过期

Redis 缓存雪崩多个：多个缓存数据在同一时间过期，导致大量请求同时访问数据库，造成数据库负载激增。

* 双缓存策略：主缓存设置短过期时间，备份缓存设置较长过期时间
* 随机过期时间

RDB和AOF的优缺点及混合持久化机制？

RDB快照：更快、文件体积小（便于传输）

AOF追加写：更完整、文本格式（便于人工检查修复）

Redis7混合持久化机制：先用RDB生成快照，然后追加AOF日志

Redis Cluster的slot分配算法？

Redis Cluster将整个键空间划分为16384个哈希槽，每个槽对应一个范围。

分配算法**：​**

1. ​对每个键计算CRC16哈希值，然后对16384取模，得到对应的slot编号。
   1. **适配65536的1/4，让哈希值均匀映射到槽**
   2. 优化心跳包的网络传输（16384b大约2KB）在节点之间传递关于哈希槽分配的信息时，用较少的比特位就可以表示一个哈希槽的状态
2. ​每个节点负责一部分slot，节点增减时重新分配，确保负载均衡。

大Key问题如何定位和解决？

1. **拆分**
2. gzip**压缩**
3. 冷数据存储到**磁盘**

热Key问题的解决方案？

1. ​**拆分**到多个节点：通过哈希算法或引入随机前缀。
2. **缓存**：CDN或本地缓存
3. ​**读写分离**：将读请求分散到从节点，减轻主节点压力。
4. ​**限流降级**

布隆过滤器的实现原理及误差率计算？

布隆过滤器是一个初始全0的m位数组、k个哈希函数、n个元素映射置1。

查询元素时任意位置为0一定不存在



Redis事务和Lua脚本的原子性区别？

Redis事务

* 通过**MULTI、EXEC、DISCARD**批量执行
* 事务执行期间其他客户端命令可能插入执行
* 如果某条命令失败后续命令仍会执行，**不支持回滚**

Lua脚本​

* 通过EVAL或EVALSHA真正完全原子
* 执行期间其他客户端命令会被阻塞
* 如果Lua 脚本中有错误直接抛出错误不会执行，**隐式“回滚”**

Java并发

CAS的原理是什么？存在哪些缺陷？

**CAS（Compare And Swap）**由 CPU 硬件指令保证原子性的无锁并发​：

比较当前值（V）与预期值（E），相等则将新值（N）写入，否则不操作。

1. ​ABA 问题：值从 A 变为 B 再变回 A，CAS 无法感知中间变化

* ​解决：版本号或时间戳

1. ​自旋开销：如果 CAS 失败，线程会不断自旋重试、消耗 CPU

* ​解决：限制自旋次数。

1. ​单变量限制：无法直接支持多个变量的原子操作。

* ​解决：封装为复合操作。

AQS的核心机制是什么？举一个基于AQS实现的工具类。

AQS（Abstract Queued Synchronizer）抽象队列式锁

核心机制：

1. ​通过 state表示**资源状态（ReentrantLock通过 state 记录锁的重入次数）**
2. ​使用​CLH 队列（双向链表）​实现自旋锁的公平排队。

* 在 CLH 队列中，线程会按照请求顺序排队，因此后面请求锁的线程不会抢先获得锁。线程会自旋等待，直到它前面的线程释放锁，这就保证了公平性。
* 通过 CLH 队列，线程被强制按照队列顺序获取锁，避免了线程饥饿的现象，保证了每个线程都有机会在合适的时间获得锁。

1. 线程通过 acquire() 和 release() 方法进入或离开队列。

提供 tryAcquire() 和 tryRelease() 等模板方法，由子类实现具体逻辑。

ConcurrentHashMap如何实现线程安全？Java7和Java8的区别？

​

Java7 分段锁（默认16个段）：锁粒度大（段级别），性能低

Java8 CAS + synchronized：锁粒度小（桶级别），性能高

synchronized和ReentrantLock的性能对比？

**synchronized**：

JVM内置优化，适合低竞争

隐式锁 自动获取释放

**ReentrantLock**：

基于AQS，适合高竞争

显式锁 手动获取释放

支持中断锁、尝试锁等特性

为什么ConcurrentHashMap的size()方法需要特殊处理？

1. 并发更新导致的不准确
2. 直接遍历的性能开销

分段累加、多次统计：  
分别统计每个 Segment 的元素数量再累加。  
通过多次统计，如果结果一致则返回，否则重试。

SpringBoot / SSM

Spring AOP的实现原理？JDK动态代理和CGLIB的区别？

Spring AOP​通过动态代理实现：

* JDK动态代理​
  + 基于接口、通过 Proxy 类生成代理对象
  + 无法代理未实现接口的类
* CGLIB代理​
  + 基于继承、通过字节码生成子类代理
  + 无法代理 final 类或方法

如何解决循环依赖问题？三级缓存机制详解。

循环依赖**：**两个或多个Bean相互依赖，导致无法完成初始化。

三级缓存**：**

1. ​一级缓存（Singleton Objects Map 单例对象）：  
   存放已经实例化、属性注入、初始化的 Bean。
2. 二级缓存（Early Singleton Objects Map 早期对象）：  
   存放已实例化，但未完成属性注入和初始化的 Bean。  
   防止 AOP每次调用ObjectFactory.getObject()都产生新代理对象。
3. 三级缓存（Singleton Factories Map 单例工厂）：  
   ObjectFactory的getObject()调用getEarlyBeanReference()创建早期Bean。

**解决循环依赖的**流程**：**

1. ​**Bean实例化**：

* 创建Bean实例，但未注入属性和初始化。
* 将Bean工厂对象（ObjectFactory）放入三级缓存。

1. ​**属性注入**：

* 发现依赖的Bean，尝试从一级缓存获取。
* 如果一级缓存不存在，从二级缓存获取。
* 如果二级缓存不存在，从三级缓存获取工厂对象，将这个 ObjectFactory 从三级缓存中移除，生成的Bean并放入二级缓存。

1. ​**Bean初始化**：

* 完成属性注入和初始化。
* 将Bean从二级缓存移除，放入一级缓存。

【4/5】Spring Bean的生命周期和作用域有哪些？

**Bean生命周期：**

1. ​实例化
2. ​属性赋值
3. ​初始化

* ​**前置处理**postProcessBeforeInitialization()。
* ​Initialization**()**：
  + @PostConstruct注解方法
  + InitializingBean的afterPropertiesSet()
  + init-method
* ​**后置处理**postProcessAfterInitialization()

1. ​使用
2. ​销毁

* @PreDestroy注解方法
* DisposableBean的destroy()
* destroy-method

**Bean的作用域：**

Singleton 默认单例

Prototype 每次请求创建一个新的Bean实例。

Request 每个HTTP请求创建一个Bean实例

Session 每个HTTP会话创建一个Bean实例

Application 每个ServletContext生命周期内创建一个Bean实例

WebSocket 每个WebSocket会话创建一个Bean实例

【4/5】SpringMVC请求处理流程中的HandlerAdapter作用？

1. DispatcherServlet接收请求
2. HandlerMapping查找处理器  
   根据请求URL找到对应Controller
3. HandlerAdapter适配并调用处理器
   * 1. 选择合适的适配器执行
     2. 通过handle()方法执行业务逻辑
4. Controller   
   执行业务逻辑，返回ModelAndView或数据对象
5. HandlerAdapter处理返回值：  
   将ModelAndView或数据对象转换为适合视图解析器的格式。
6. DispatcherServlet返回响应  
   将处理后的渲染视图或JSON数据返回给客户端。

【3/4】SpringBoot自动配置的实现机制（@EnableAutoConfiguration）？

1. ​@SpringBootApplication =   
    @SpringBootConfiguration

+ @EnableAutoConfiguration // 关键：开启自动配置

+ @ComponentScan  
Spring Boot启动时，AutoConfigurationImportSelector 通过 SpringFactoriesLoader 机制，加载 META-INF/spring.factories 中定义的 自动配置类。扫描META-INF/spring.factories，加载自动配置类。

1. ​条件匹配@Conditional：根据类路径、配置文件、Bean是否存在判断是否加载配置类。
   1. @ConditionalOnClass
   2. @ConditionalOnMissingBean

【3/4】Spring事务失效的常见场景有哪些？

1. ​异常类型不匹配  
   默认只回滚RuntimeException和Error，如果抛出比如IOException且未配置rollbackFor，不会回滚。
2. ​异常没抛出
3. ​同类方法调用  
   同一个类中，方法A调用方法B（B有@Transactional），实际调用的是原始对象的方法，而不是代理对象的方法，而事务管理是通过代理机制实现的，B的事务不会生效。
4. ​非public方法、final方法、static 方法
5. ​多线程调用：事务基于ThreadLocal实现，多线程事务无法传递。
6. ​其他
   * **事务传播行为**PROPAGATION\_NOT\_SUPPORTED会挂起事务
   * 未正确配置数据源DataSourceTransactionManager
   * MyISAM引擎本身不支持事务
   * 非Spring管理Bean

【4/4】MyBatis的#和$的区别及SQL注入问题。

* #：预编译、更安全
* $：直接拼接、存在SQL注入风险

动态表名、排序字段必须用$。因为表名、字段名、ORDER BY 排序字段等 SQL 结构 不能使用 ? 作为参数绑定。只有值（变量）可以用 # 作为参数占位符。

由于 $ 存在 SQL 注入风险，我们需要严格控制用户输入，确保只能传递合法的表名和字段名（白名单校验/使用 Enum 限制可选字段）

【3/3】SpringBoot如何集成Tomcat容器？

默认内嵌Tomcat容器

* **端口**：8080
* **最大连接数**：10000  
  TCP 连接的数量，即有多少个客户端可以连接到服务器。
* **最大线程数**：200  
  可用的工作线程，表示服务器可以同时处理多少个请求。

计算机网络

【3/5】TCP三次握手的详细过程和状态变化？

三次握手

* 第一次: 客户端发出同步请求报文。
* 第二次：服务器发出同步确认报文。
* 第三次: 客户端发出确认报文。



四次挥手

* 第一次：客户端发出连接释放报文。
* 第二次：服务器发出确认报文。
* 第三次：服务器发出连接释放确认报文。
* 第四次：客户端发出确认报文。



**TIME\_WAIT状态的作用：​**

1. 防止旧连接的延迟数据包干扰新连接。
2. 保证对端收到最后的ACK，避免连接关闭异常。

**TIME\_WAIT过多的解决方案：​**

1. 调整内核参数：减少net.ipv4.tcp\_tw\_reuse
2. 增加可用端口范围：扩大net.ipv4.ip\_local\_port\_range。
3. 使用连接池复用连接，减少频繁创建和关闭。
4. 减少短连接的使用。

【4/5】HTTPS和HTTP有哪些区别？

**1**. 端口号

* **HTTP**：默认为80端口。
* **HTTPS**：默认为443端口。

2. 加密

* **HTTP**：数据在传输过程中是未加密的，容易被拦截、篡改。
* **HTTPS**：通过SSL/TLS协议进行加密，使得数据在传输过程中即使被截获，也无法解密。

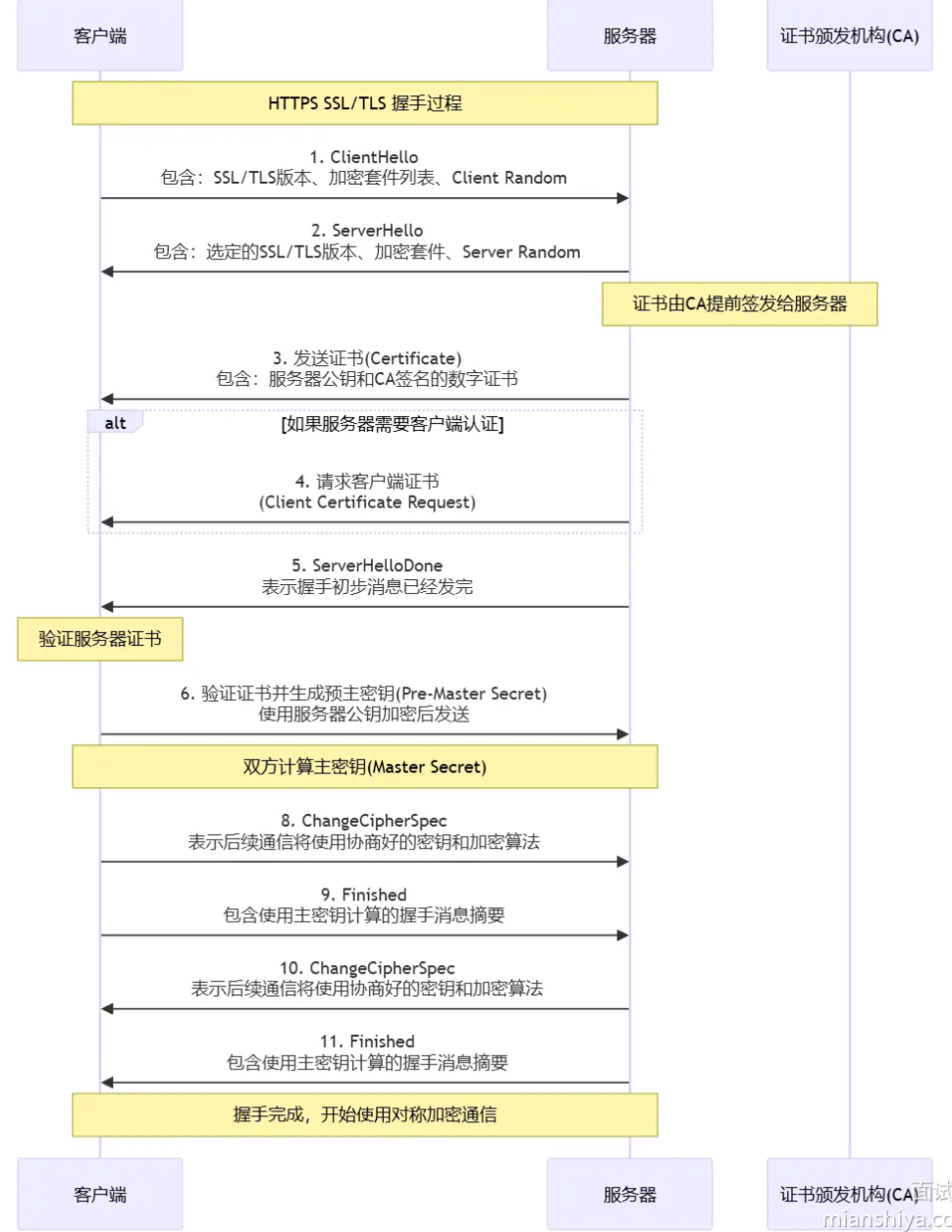
【3/5】HTTPS工作原理是什么？它是如何实现数据加密的？

1. SSL/TLS协议

* 对称加密、非对称加密、哈希算法的结合。

2. SSL/TLS握手过程

1. **客户端Hello**：客户端向服务器发送支持的加密算法、随机数。
2. **服务器Hello**：选择的加密算法、随机数 + 数字证书（包含公钥）
3. **验证数字证书**
4. **密钥交换**：  
   客户端使用服务器的公钥生成会话密钥，发送给服务器。  
   服务器使用自己的私钥解密，获得会话密钥。
5. **加密通信**。



【4/5】TCP快速重传和超时重传的区别？

**快速重传：**

1. **触发条件：**收到3个重复ACK（表明后续数据包丢失）。**响应**更快
2. 适用于部分数据包丢失但网络状况较好时。

**超时重传：**

1. **触发条件：**未收到ACK且重传计时器超时。**响应**较慢
2. 适用于严重丢包或连接中断。

【3/5】QUIC协议如何解决队头阻塞问题？

1. ​基于UDP：​避免了TCP层的队头阻塞。
2. ​多路复用 + 独立流量控制和快速重传

【4/5】TCP拥塞控制？



【2/4】HTTP1.1新特性？

1. 持久连接复用同一连接

2. 请求管道化：同时发送多个请求

3. 更细粒度的缓存控制：新增Cache-Control头部

4. Host头部：允许同一IP托管多个虚拟主机（即多个域名）

5. 新增错误代码：410资源已永久删除

【2/4】HTTP2.0新特性？

1. 多路复用应用层消除队头阻塞

2. 二进制协议以帧传输

* **1.1是基于文本的协议**

3. 独立流量控制

* **1.1主要依赖TCP的拥塞控制，可能导致带宽浪费。**

4. 头部压缩（HPACK算法）

* **1.1头部信息未压缩，且请求头较为冗长，这增加了网络带宽的消耗，尤其对于重复的请求头，如Cookies等，效果显著。**

5. 服务器主动推送

【2/4】HTTP3.0新特性？

1. 基于QUIC协议

* **1和 2**依赖TCP协议
* Quick UDP Internet Connections是Google提出，建立在UDP上

2. 零RTT连接建立

* **1和2**基于TCP的三次握手
* **3**客户端在首次发送请求时，可以立即开始数据传输

3. 传输层消除队头阻塞

* **HTTP2**虽然支持多路复用，但仍然受TCP队头阻塞的影响。当一个流的数据包丢失时，整个连接的延迟会增加，影响其他流的传输。

4. 将TLS加密内建于协议

操作系统

【3/5】进程和线程的区别？协程的优势是什么？

进程：是资源分配的基本单位，一个进程可以包含多个线程

进程有独立的代码、数据、文件等，创建和切换开销大

进程之间是相互独立的，需要**进程间通信（IPC）机制**，如管道、消息队列、共享内存、套接字等

线程：是CPU调度的基本单位，属于进程

线程组内共享资源，但有独立的PC和堆栈，创建和切换开销较小

线程直接读写内存即可，但需要**同步机制**以避免数据竞争

【3/5】select/poll/epoll的区别及边缘触发模式？

1. ​**select**：

* 使用 ​fd\_set ，支持的文件描述符数量有限（通常 1024）。
* 每次调用需要遍历所有文件描述符，效率低。时间复杂度：O(n)。

1. ​**poll**：

* 使用 ​pollfd 数组，支持的文件描述符数量无限。
* 每次调用需要遍历所有文件描述符，效率低。时间复杂度：O(n)。

1. ​**epoll**：

* 使用事件驱动机制，只关注活跃的文件描述符，效率高。
* 支持边缘触发（ET）和水平触发（LT）模式。时间复杂度：O(1)。

**边缘触发模式（ET）：**

* ​只在状态变化时通知一次，需一次性处理所有数据，否则会丢失事件。
* ​**优点**：减少事件通知次数，提高效率。
* ​**适用**：高性能、高并发的网络编程。

【3/5】用户态和内核态切换的开销来源？

1. ​保存和恢复寄存器、程序计数器等状态。
2. ​权限**级别切换**
3. ​缓存**失效**
4. ​系统调用需要内核验证，增加额外时间。
5. ​切换可能伴随中断处理，进一步增加开销。

【2/5】进程间通信共享内存和消息队列的对比？

**共享内存**：

* ​多个进程映射同一块物理内存，需要同步机制（如信号量）避免竞争。
* ​速度快，无数据拷贝，适用高性能、大数据量通信。

**消息队列**：

* ​内核维护消息队列，进程发送/接收消息通信，自带同步机制
* ​速度慢，有数据拷贝，适用小数据量、异步通信

【1/5】零拷贝技术（sendfile）的实现原理？

零拷贝技术（sendfile）在文件传输中，让内核直接将文件数据从磁盘读取到网络缓冲区，无需经过用户态，避免数据在用户态和内核态之间的拷贝，提升性能。

* 传统方式：磁盘 -> 内核缓冲区 -> 用户缓冲区 -> 网络缓冲区
* sendfile ：磁盘 -> 网络缓冲区

【1/4】内存映射（memory map, mmap）系统调用的实现原理？

1. ​mmap系统调用将文件/设备映射到进程的虚拟地址空间，实现文件读写或共享内存。
2. ​**实现步骤**：

* ​**创建映射**：内核在进程的虚拟地址空间中分配一段区域，建立虚拟内存与文件/设备的映射关系。
* ​**延迟加载**：映射时不立即加载文件内容，而是通过缺页中断按需加载。
* ​**读写操作**：进程通过指针直接访问映射区域，内核负责将数据同步到文件或设备。
* ​**解除映射**：调用 munmap 时，释放虚拟地址空间并同步数据到文件。

1. **优点**：减少用户态和内核态的数据拷贝，提高性能。

【2/3】虚拟内存的作用？

* ​扩展：允许程序使用比物理内存更大的地址空间。
* ​隔离：每个进程拥有独立的虚拟地址空间，提高安全性。
* ​管理：简化内存分配和回收，支持动态内存需求。
* ​共享：多个进程可以共享同一块内存区域。

Java集合

【5/5】详细描述HashMap的扩容机制（触发条件、rehash过程）。

**Java7扩容条件：**

元素数量超过阈值 && 发生哈希冲突

**Java8扩容条件：**

1. 情况一：元素数量超过阈值
2. 情况二：链表长度≥8 && 数组长度＜64，优先扩容而不是转为红黑树

Rehash过程**：**

1. 新建一个2倍大小的数组
2. 只需部分移动：if e.hash & oldCap==0
   1. 0接在loTail后面
   2. 1接在hiTail后面

基础：

数组长度是2的次方，且扩容为2倍。

**oldCap**：

位运算代替取模提高效率、保证hash均匀分布：hash & (数组长度-1)

旧长16即010000，16-1=15即001111

新长32即100000，32-1=31即011111

rehashing时用**旧长16**即010000作为一个Mask看对应该位是否为1

源码**if e.hash & oldCap==0**（0接在loTail后面、1接在hiTail后面）

【3/3】HashMap底层结构是什么？如何解决哈希冲突？

**HashMap底层结构**是数组+链表，Java8后引入红黑树。

当链表长度≥8且 数组容量≥64时，链表会转换为红黑树  
（链表长度≤8时，链表性能足够且内存占用低）

当红黑树节点数≤6时，红黑树会退化为链表

**解决哈希冲突**：

1. **拉链法**
2. **红黑树**
3. **二次扰动函数：**如果哈希值低位变化较小，计算索引时，低位相同的键会映射到同一位置。**将哈希值高位低位异或、优化索引分布、减少哈希冲突**
4. **动态扩容机制：**
   1. 超过阈值（容量×负载因子，默认0.75）扩容2倍并重新计算索引。
   2. 0.75是空间利用与哈希冲突的最佳平衡  
      负载因子过高会增加哈希冲突  
      **负载因子**过低则导致频繁扩容

【3/3】为什么HashMap线程不安全？举例说明并发问题场景。

**HashMap线程不安全的原因：**

数组数据覆盖、头插环形链表、红黑树树化结构异常

**解决方案：**

1. Collections.synchronizedMap(new HashMap<>())：在方法级别加锁，性能低
2. ConcurrentHashMap：采用分段锁（7）或CAS+ synchronized（8），性能高。

【2/2】HashMap的遍历方式有哪几种？哪种效率更高？

1. ​entrySet() 效率最高、推荐使用。
2. forEach()+ Lambda 底层实现基于entrySet()。
3. ​keySet() 返回所有键的集合，通过 map.get(key)获取值
4. values() 返回所有值的集合。由于无法通过值获取键

for (Map.Entry<String, Integer> entry : map.entrySet()) {

String key = entry.getKey();

Integer value = entry.getValue();

System.out.println(key + ": " + value);

}

map.forEach((key, value) -> {

System.out.println(key + ": " + value);

});

Linux命令

【3/5】如何用grep查找包含"error"的日志并统计次数？

* grep "error" logfile.log：查找 logfile.log 中包含 "error" 的所有行
* grep -c "error" logfile.log：统计 logfile.log 中包含 "error" 的行数

【3/5】awk如何实现按列求和？

awk '{sum+=$3} END{print sum}' filename

* {sum+=$3}：逐行累加第3列的值。
* END{print sum}：处理完所有行后，输出累加结果。

【3/5】如何用sed批量替换文件中的字符串？

sed -i 's/old\_string/new\_string/g' filename

* sed：Stream Editor 流编辑器
* -i：in-place，直接原地修改文件、不输出到终端
* s：**替换命令**，格式为 s/查找内容/替换内容/
* g ：全局替换

【2/5】strace和perf工具的作用及使用场景？

strace：跟踪进程系统调用

strace -p <pid> 监视进程 <pid> 的所有系统调用。

perf：分析 CPU 性能瓶颈、函数调用链、硬件事件

perf top 实时查看占用 CPU 最高的函数

【2/5】如何用tcpdump抓取指定端口的SYN包？

tcpdump -i eth0 'tcp[tcpflags] & tcp-syn != 0 and port 80'

* -i eth0：指定网卡接口
* tcp[tcpflags] & tcp-syn != 0：匹配 SYN标志位
* port 80：指定端口